Федеральное государственное автономное
образовательное учреждение высшего образования

Университет ИТМО

Дисциплина: Архитектура программных систем

Лабораторная работа 2

Выполнил:
Румский А.М.

Группа: P33121

2023 г.

г. Санкт-Петербург

GOF паттерны:

Builder: шаблон, который отделяет построение сложного объекта от его представления, позволяя одному и тому же процессу построения создавать различные представления.

Пример 1: редактор документов, в котором можно создавать документы в различных форматах, таких как txt, HTML и Markdown. Шаблон конструктора можно использовать для создания этих документов шаг за шагом. Класс Director может определять порядок, в котором создаются элементы документа (текст, изображения, таблицы), а различные классы ConcreteBuilder могут реализовывать процесс построения для каждого формата документа.

```
//Builder interface
interface DocumentBuilder {
    void buildHeader(String text);
    void buildParagraph(String text);
    void buildImage(String path);
    //...

    Document getResult();
}

//ConcreteBuilder for txt documents
class TxtDocumentBuilder implements DocumentBuilder {
    //implementation  ...
}
//ConcreteBuilder for HTML documents
class HTMLDocumentBuilder implements DocumentBuilder {
    //implementation  ...
}

//ConcreteBuilder for Markdown documents
class MarkdownDocumentBuilder implements DocumentBuilder {
    //implementation  ...
}

//Director class
class DocumentDirector {
    private DocumentBuilder builder;

    public DocumentDirector(DocumentBuilder builder) {
        this.builder = builder;
    }

    public void construct() {
        builder.buildHeader("Sample Document");
        builder.buildParagraph("This is a paragraph.");
        builder.buildImage("/path/to/image.png");
        //...
    }
}

//Product class
class Document {
    //document representation
}
```

Пример 2: создание объекта конфигурации для системы с различными параметрами конфигурации. Значения по умолчанию для необязательных параметров подставляются автоматически.

```
//Builder interface
interface ConfigurationBuilder {
    void setOption1(String value);
    void setOption2(int value);
    void setOption3(boolean value);
    //...

    Configuration getResult();
}

//ConcreteBuilder
class DefaultConfigurationBuilder implements ConfigurationBuilder {
    //implementation...
}

// Director class
class ConfigurationDirector {
    private ConfigurationBuilder builder;

    public ConfigurationDirector(ConfigurationBuilder builder) {
        this.builder = builder;
    }

    public void construct() {
        builder.setOption1("Default");
        builder.setOption2(42);
        //...
    }
}

//Product class
class Configuration {
    //configuration
}
```

Ограничения:

Неприменим к простым случаям – будет являться чрезмерным усложнением.

Повышение сложности кода из-за создания множества классов.

Во многих реализациях шаблона Builder созданные объекты часто являются неизменяемыми.

Adapter: шаблон проектирования, который позволяет несовместимым интерфейсам работать вместе.

Пример 1: пусть есть существующее приложение, использующее платформу ведения журнала с определенным интерфейсом. Для переключения на новую платформу с другим интерфейсом, вместо изменения всех классов, использующих старую платформу, можно создать адаптер.

```
//Existing logging framework
interface Logger {
    void log(String message);
}

//New logging framework
class NewLogger {
    void logMessage(String msg) {
        System.out.println("New Logging: " + msg);
    }
}

//Adapter
class LoggerAdapter implements Logger {
    private NewLogger newLogger;

    public LoggerAdapter(NewLogger newLogger) {
        this.newLogger = newLogger;
    }

    @Override
    public void log(String message) {
        newLogger.logMessage(message);
    }
}

//Client
public class Client {
    public static void main(String[] args) {
        Logger logger = new LoggerAdapter(new NewLogger());
        logger.log("Hi from new platform.");
    }
}
```

Пример 2: есть интерфейс для компонента GUI со списком методов, но вы хотите реализовать только несколько из этих методов для определенного компонента.

```java
//Target interface with multiple methods
interface GUIComponent {
    void draw();
    void resize();
    void moveTo(int x, int y);
}

//Interface Adapter (provides default implementations for some methods)
abstract class InterfaceAdapter implements GUIComponent {
    @Override
    public void draw() {
        //Default implementation
    }

    @Override
    public void resize() {
        //Default implementation
    }

    @Override
    public void moveTo(int x, int y) {
        //Default implementation
    }
}

//Concrete class implementing only a subset of methods
class SimpleButton extends InterfaceAdapter {
    @Override
    public void draw() {
        //Implementation
    }
}

//Client code
public class Client {
    public static void main(String[] args) {
        GUIComponent button = new SimpleButton();
        button.draw();
    }
}
```

Ограничения:

Использование адаптеров может привести к увеличению времени отклика программы из-за дополнительной обработки команд.

Шаблон не следует использовать в качестве обходного пути для проблем со структурой программы.

Шаблон является однонаправленным, что означает, что он адаптирует интерфейс адаптируемого к целевому объекту.

Если класс адаптируемого объекта изменяется, может потребоваться модификация адаптера.

Не всегда адаптер является единственным вариантом решения проблемы, необходимо рассматривать и другие шаблоны (например Bridge).

GRASP паттерны:

Controller: отвечает за обработку входных системных событий, делегируя обязанности по их обработке компетентным классам.

Пример 1: не уверен подходит ли, но - класс из 4 лабораторной по Web-программированию, который обрабатывает запросы с разных адресов, и в соответствии с этим вызывает методы, внутри которых данные обрабатываются отдельными классами.

```
@Transactional
@CrossOrigin
@RestController
@RequestMapping("/main")
public class Controller {

    @Autowired
    HitRepository hitRepository;
    @Autowired
    AuthRepository authRepository;

    @GetMapping("/hits/{login}" )
    public ResponseEntity<List<HitEntry>>
getAllHitEntriesByLogin(@PathVariable("login") String login){
        try {
        //reques to the responsible class
            return new ResponseEntity<>(entries, HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(null,
HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    @PostMapping("/authentication")
    public ResponseEntity<AuthEntry> authUser(@RequestBody AuthEntry
entry){
        AuthEntry dbEntry = authRepository.findByLogin(entry.getLogin());
        if((dbEntry !=null &&
dbEntry.getPassword().equals(Encryptor.getSHA512Encode(entry.getPassword()
)) && !dbEntry.isActive())){
        //reques to the responsible class
        }else{
            return new ResponseEntity<>(null,HttpStatus.NOT_FOUND);
        }
    }

    @PostMapping("/registration")
    public ResponseEntity<AuthEntry> registerUser(@RequestBody AuthEntry
entry){
        //reques to the responsible class
            return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
        }
    }

    @PostMapping("/logout")
    public ResponseEntity<AuthEntry> logout(@RequestBody AuthEntry entry){
        //reques to the responsible class
        return new ResponseEntity<>(null, HttpStatus.OK);
    }
    @PostMapping("/hits/add")
    public ResponseEntity<HitEntry> addEmployee(@RequestBody HitEntry hit)
{
        //reques to the responsible class
        return new ResponseEntity<>(null, HttpStatus.OK);
    }

    @PostMapping("/hits/clear")
    public ResponseEntity<?> deleteEmployee(@RequestBody AuthEntry entry)
{
        //reques to the responsible class
        return new ResponseEntity<>(null,HttpStatus.OK);}
```

Пример 2: Система выдачи и регистрации книг/мониторинга дат сдачи/ в библиотеке.

LibraryController выполняет регистрацию и выдачу книг, управление просроченными штрафами и обработку запросов на бронирование книг. Так же он отвечает за координацию с BookService для обновления статуса книг, взаимодействие с UserService для управления учетными записями пользователей и штрафами, а также за обновление пользовательского интерфейса соответствующей информацией.

```
public class BookService {
    public void updateBookStatus(String bookId, String status) {
        // Code to update the status of a book in the database
    }
}

public class UserService {
    public void manageOverdueFines(String userId) {
        // Code to manage overdue fines for a user
    }
}

public class LibraryController {
    private BookService bookService;
    private UserService userService;

    public LibraryController(BookService bookService, UserService
userService) {
        this.bookService = bookService;
        this.userService = userService;
    }

    public void checkOutBook(String userId, String bookId) {
        // Code to handle book check-out
        bookService.updateBookStatus(bookId, "Checked Out");
        userService.manageOverdueFines(userId);
    }
}
```

Ограничения:

Шаблон часто подразумевает сильную связь с UI, что означает, что они тесно связаны с компонентами UI.

Шаблон может создать единую точку отказа, если контроллер становится ответственным за слишком много критически важных функций.

Контроллеры могут быть тесно связаны с определенной структурой пользовательского интерфейса, что ограничивает их повторное использование в разных контекстах.

В больших системах контроллер может стать слишком большим и сложным в управлении. Следовательно, необходимо создавать контроллеры для определенного списка задач и не перегружать их.

Indirection: реализует низкую связность между классами, путем назначения обязанностей по их взаимодействию дополнительному объекту — посреднику.

Пример 1: Контроллер из модели MVC для Web-приложений, который перенаправляет запросы в соответствии с условием.

```
@WebServlet(name = "controller", value = "/controller")
public class ControllerServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        RequestDispatcher requestDispatcher =
req.getRequestDispatcher("url");
        requestDispatcher.forward(req, resp);
        String attr = req.getAttribute(...);
        if(con){
        //logic
        requestDispatcher.forward(req, resp);
        }
        else if(con1){
        //logic
        requestDispatcher.include(req, resp);
        }
        else if(con2){
        //logic
        resp.getWriter().out.println(...);
        }
        //...
    }
}
```

Пример 2: пусть есть файловый менеджер, в котором можно взаимодействовать с файлами(неожиданно!). Тогда FileGateway можно использовать как промежуточный элемент, спомощью которого расширить базовые возможности менеджера (шифрование файлов, их валидация и т.д.).

```
public class FileManager {
    public void createFile(String fileName) {
        //Code to create a file
    }
}

//Indirection
public class FileGateway {
    private FileManager fileManager;

    public FileGateway(FileManager fileManager) {
        this.fileManager = fileManager;
    }

    public void createFile(String fileName) {//Additional logic}
}
```

Ограничения:

Шаблон не стоит использовать в ситуациях, когда программа достаточно проста, чтобы обойтись без него, так как в противном случае это может привести к чрезмерному услужению кода как минимум, и к замедлению программы в целом как максимум.

Вывод: повторно ознакомился с паттернами проектирования GRASP & GOF.